Lattice QCD GPU programming with Quda and Grid

October 28, 2025

renwick.james.hudspith@gmail.com

Outline

- Writing an interface for Quda
- Writing a simple kernel
- Batched BLAS calls with cuBLAS

What we learned last lecture

Graphics hardware was too niche in the past to be useful but now we live in the era of General Purpose GPU programming.

We learned the kernel runs threadblocks, each with some number of Warps which are bundles of threads on a Streaming Multiprocessor (SM).

A warp in CUDA is 32 threads and this will always be run in a SIMD way. Thread and block parallelism is up to the user in CUDA and is set at kernel launch by the triple-chevron `<<<block,thread>>>`

We need to push stuff from the Host (CPU) to the Device (GPU) via the PCIE port and this can still be costly.

QUDA and Grid approach GPU parallelism differently.

Results from the homework (2^{24} floats) vadd and vsum on Perlmutter

vadd

Execution	time (ms)
1 Thread	0.284
2 Thread	0.004
4 Thread	0.003
8 Thread	0.002
16 Thread	0.002
32 Thread	0.002
64 Thread	0.002
256 Thread	0.002
Copy	10.6+5.2

vsum

Execution	time (ms)
1 Thread	0.291
2 Thread	0.030
4 Thread	0.029
8 Thread	0.016
16 Thread	0.010
32 Thread	0.008
64 Thread	0.008
256 Thread	0.007
Copy	9.9+Block

Simple operations are completely limited by copying to device (worse with malloc, used cudaHostMalloc). vadd saturates with few threads, vsum is more complicated.

Jamie's Laptop (i3-1215U 4 physical cores) OpenMP
vadd: 15ms — vsum: 4.5ms

renwick.james.hudspith@gmail.com

What is a Quda?

Quda (lattice QCD on CUDA https://github.com/lattice/quda) is a templated c++ library that under the hood interacts with the C-API of CUDA, or HIP, or SYCL.

It is quite heavily templated so that lots of operations such as quark propagator inversions in mixed precision (double/single/half/quarter) by a variety of algorithms are possible. It is to be compiled with cmake (or even better with the ccmake GUI). nvcc is a slow compiler and with all the templating Quda takes an astonishingly long time to compile. This makes development in it painful.

Personally, I think there are too many options in Quda. The library is quite mature and it suffers from feature creep and bloated inputs. I also wish it had a release schedule - it used to. The develop branch changes the interface quite often.

Generic interfaces

Quda provides convenient wrappers for CUDA/HIP/SYCL calls

```
cudaMalloc -> pool_device_malloc

cudaMemcopy -> qudaMemcopy

cudaFree -> pool_device_free
```

A simple code that copies an integer to the device and sends it back is:

```
void do_very_little( void )
{
  int a = 42 , b = 0 ;
  void *d_tmp = pool_device_malloc(sizeof(int));
  qudaMemcpy(d_tmp, &a, sizeof(int), qudaMemcpyHostToDevice);
  qudaMemcpy(&b, d_tmp, sizeof(int), qudaMemcpyDeviceToHost);
  pool_device_free(d_tmp);
  printf( "A %d :: B %d\n" , a , b ) ;
}
```

ColorSpinor interface

I think modern QUDA doesn't want us to interface with the GPU like this, instead it provides a convenient object the ColorSpinor. Let's look at that instead, first we need to instantiate some params using a constructor:

```
ColorSpinorParam cpu_evec_param( host_evec, inv_param, x,
false, QUDA_CPU_FIELD_LOCATION );
cpu_evec_param.nSpin = 1;
```

Here we tell QUDA we have an array of color-vectors on the host called `host_evec`, of course we need to pass some `inv_params` to it because this is QUDA and inverter params are passed everywhere. Here "x" are our local dimensions.

Now let's create a vector of these guys on the host

```
std::vector<ColorSpinorField> evec(nEv);
for (int iEv=0; iEv<nEv; ++iEv) {
  cpu_evec_param.v = host_evec[iEv];
  evec[iEv] = ColorSpinorField(cpu_evec_param) ;
}
```

The ".v" is a pointer to the memory address of `host_evec`.
From here we can create the same on the device (setting the internal precisions appropriately)

```
ColorSpinorParam cuda_evec_param( cpu_evec_param,
            inv_param, QUDA_CUDA_FIELD_LOCATION );
cuda_evec_param.setPrecision( inv_param.cuda_prec,
  inv_param.cuda_prec, true );
std::vector<ColorSpinorField> quda_evec(1) ;
quda_evec[0] = ColorSpinorField(cuda_evec_param);
```

Great, so we have one eigenvector on the device and a bunch of pointers to those on the host. We can now equate these

```
for (int i=0; i<nEv; i++) {
   quda_evec[0] = evec[i] ;
}
```

And QUDA does all the necessary copies. We do not need to free these objects as they are destructed when they fall out of scope.

Let's do something nice an simple

Ok, so we're going to use the ColorSpinorFields to do something, the trace of a product i.e. for a pair of color fields x_a and y_a over all positions "i" we will do

$$R(i) = \sum_a x_a^\dagger(i) y_a(i) \tag{1}$$

i.e. $< x|y >$ we will call this InnerProduct. This requires very simple parallelism like in the CUDA examples over vectors from the last lecture. It all starts with a .cu file we will call "`color_contract.cu`", this will live in "/quda/lib/" and will be added to the targets in "quda/lib/CMakeLists.txt" with all the other .cu files.

So what is in this file? Well there are some header files

```
#include <color_spinor_field.h>
#include <kernels/color_contract.cuh>
#include <contract_quda.h>
#include <tunable_nd.h>
#include <tunable_reduction.h>
#include <instantiate.h>
```

It expects a kernel header file "/quda/include/kernels/" called "`color_contract.cuh`". Which we will get back to. And some general cruft that defines the objects we will use.

Continuing on we open the quda namespace and declare our template, which inherits `TunableKernel2D`

```
namespace quda {
  // Inner Product
template <typename Float, int nColor> \
  class InnerProduct : TunableKernel2D
  {
    protected:
    const ColorSpinorField &x;
    const ColorSpinorField &y;
    complex<Float> *result;
    unsigned int minThreads() const { return x.VolumeCB(); }
```

Here Float is templated and nColor is passed. ColorSpinorFields x and y are shallow copied and we have a result pointer. We need to set minThreads to be VolumeCB() - the single-parity volume.

Here we have our public methods as it is extendening TunableKernel2D it has a launch method

```
public:
  InnerProduct(const ColorSpinorField &x,
      const ColorSpinorField &y,
      void *result) :
    TunableKernel2D(x, 2),
    x(x),
    y(y),
    result(static_cast<complex<Float>*>(result))
  {
    apply(device::get_default_stream());
  }
```

```
void apply(const qudaStream_t &stream)
{
  TuneParam tp = tuneLaunch(*this, getTuning(),
                            getVerbosity());
  ColorContractArg<Float, nColor> arg(x, y, result);
  launch<InnerProd>(tp, stream, arg);
}
}
```

And we finally define what "apply" means by creating a TuneParam object, defining "arg" to be of type ColorContractArg (from the header) and finally launching the kernel InnerProd. So far this does nothing.

We now define the outward-facing function we will call, called "innerProductQuda"

```
void innerProductQuda(const ColorSpinorField &x,
                      const ColorSpinorField &y,
                      void *result)
{
  checkPrecision(x, y);
  if (x.Nspin() != 1 || y.Nspin() != 1) {
    errorQuda("Unexpected number of spins x=%d y=%d",
     x.Nspin(), y.Nspin());
   }
  instantiate<InnerProduct>(x, y, result);
}
```

This does some sanity checks that x and y are the same size and instantiates the InnerProduct method, which as we have seen launches the "InnerProd" kernel.

Ok so we are most of the way there! We just need to know what is in this header "`#include <kernels/color_contract.cuh>`" the actual kernel.

```
#include <color_spinor_field_order.h>
#include <index_helper.cuh>
#include <quda_matrix.h>
#include <matrix_field.h>
#include <constant_kernel_arg.h>
#include <kernel.h>
```

```
namespace quda {
    template <typename Arg> struct InnerProd {
    const Arg &arg;
    constexpr InnerProd(const Arg &arg) : arg(arg) {}
    static constexpr const char *filename() {
        return KERNEL_FILE; }
    __device__ __host__ inline void operator()(int x_cb,
                                               int parity)
    {
      using real = typename Arg::real;
      using Vector = ColorSpinor<real, Arg::nColor, Arg::nSpin>;
      Vector x = arg.x(x_cb, parity);
      Vector y = arg.y(x_cb, parity);
      arg.s[x_cb + parity*arg.threads.x] = \
            innerProduct(x, y, 0, 0) ;
    }
  };
```

```
template <typename Float, int nColor_> struct ColorContractArg :
       kernel_param<> {
    using real = typename mapper<Float>::type;
    static constexpr int nColor = nColor_, nSpin = 1;
    // Create a typename F for the ColorSpinorFields
    typedef typename colorspinor_mapper<Float, nSpin, nColor, \
     false, false>::type F;
    F x, y ;
    complex<Float> *s;
    dim3 threads;     // number of active threads required
    int_fastdiv X[4]; // grid dimensions
    ColorContractArg(const ColorSpinorField &x,
    const ColorSpinorField &y, complex<Float> *s) :
      x(x), y(y),s(s), threads(x.VolumeCB())
    {
      for(int i=0; i<4; i++) { X[i] = x.X()[i]; }
    }
  };
```

And there it is. The kernel calls an inlined function called innerProduct

It is worth noting some things here. x and y are split into `x_cb` and parity. Where `x_cb` is the checkerboarded volume and parity is 0 or 1. These are mapped to the threadIdx.x and threadIdx.y of the 2D kernel mapping in the `TunableKernel2D(x,2)` instantiation.

innerProduct over color and spin is actually to be found on line 965 of "`/include/color_spinor.h`" on the next slide.

```
  template <typename Float, int Nc, int Nsa, int Nsb>
  __device__ __host__ inline complex<Float> innerProduct(
  const ColorSpinor<Float, Nc, Nsa> &a,
  const ColorSpinor<Float, Nc, Nsb> &b, int sa, int sb)
  {
    complex<Float> dot = cmul(conj(a(sa, 0)), b(sb, 0));
#pragma unroll
    for (int c = 1; c < Nc; c++) {
      dot = cmac(conj(a(sa, c)), b(sb, c), dot);
    }
    return dot;
  }
```

Lots of these ColorSpinor objects have their own inlined kernels that we can just call within our own kernel header file.

This is a simple kernel under the hood but the implementation defines

- innerProductQuda - the actual function we will call (`color_contract.cu`)
- InnerProduct - the class that extends TunableKernel2D launching the kernel "InnerProd" (`color_contract.cu`)
- InnerProd - the kernel function being called $\rightarrow$ calls innerProduct (`kernels/color_contract.cuh`)
- innerProduct - the actual inlined kernel function (`color_spinor.h`)

All so that we can dance around the autotuner

This is somewhat the recommended way to work (see the wiki page)

Quite a few of the kernels follow this pattern and for the most part it is boiler plate that can be copy-pasted. Of course the appropriate additions need to be made to the CMakeLists

Homework - investigating the performance of QUDA

Check out my version of QUDA https://github.com/RJHudspith/quda which has the Color trace code in.

If you checkout my branch of `quda_laph` `https://github.com/cosmon-collaboration/quda_laph/tree/ccmake_co` there is a test file that will be built with the library in `tests/test_ColorContract.cc`. This code can compute the color trace on the CPU and on the GPU.

Extend this code to do N traces and compare CPU and GPU performance. Where is the "break even point"?

Why batched BLAS?

As we have seen GPU hardware does best when you utilise the fast L1 cache per SM and can reuse data locality. For this we can use cuBLAS, the BLAS library for CUDA. Fortunately hipBLAS has the same routines and functionality, so the discussion remains the same.

cuBLAS only works in column-major (i.e. Fortran) order

cuBLAS will handle the blocking and threading, a single function call is still a kernel launch though.

A batched GEMM (with p-batches) looks like

$$C[p] = \alpha \mathrm{op}(A[p])\mathrm{op}(B[p]) + \beta C[p] \tag{2}$$

Where p is some index and op is some operation - Transpose, Conjugate Transpose or none.

And the function to call is

```
<T>gemmStridedBatched(cublasHandle_t handle,
                      cublasOperation_t transA,
                      cublasOperation_t transB,
                      int M, int N, int K,
                      const T* alpha,
                      const T* A, int ldA, int strideA,
                      const T* B, int ldB, int strideB,
                      const T* beta,
                      T* C, int ldC, int strideC,
                      int batchCount)
```

If you are familiar with BLAS calls we have the familiar "leading dimensions" of the matrices A, B, and C which allow for different parts of the matrix to be strided.

In nested loop form (column major) it looks like:

```
for(int p=0;p<batchCount;p++){
 for(int m=0;m<M;m++){
  for(int n=0;n<N;n++){
    T tmp = 0 ;
    for(int k=0;k<K;k++){
      tmp+=A[m+lda*k+strideA*p]*B[k+ldb*n+strideB*p];
    }
    C[m+n*lda+p*strideC]=\
       (*alpha)tmp+(*beta)*C[m+n*ldc+strideC*p];
  }
 }
}
```

Why do we want to do this? Small, dense, matrix performance is **much** better!

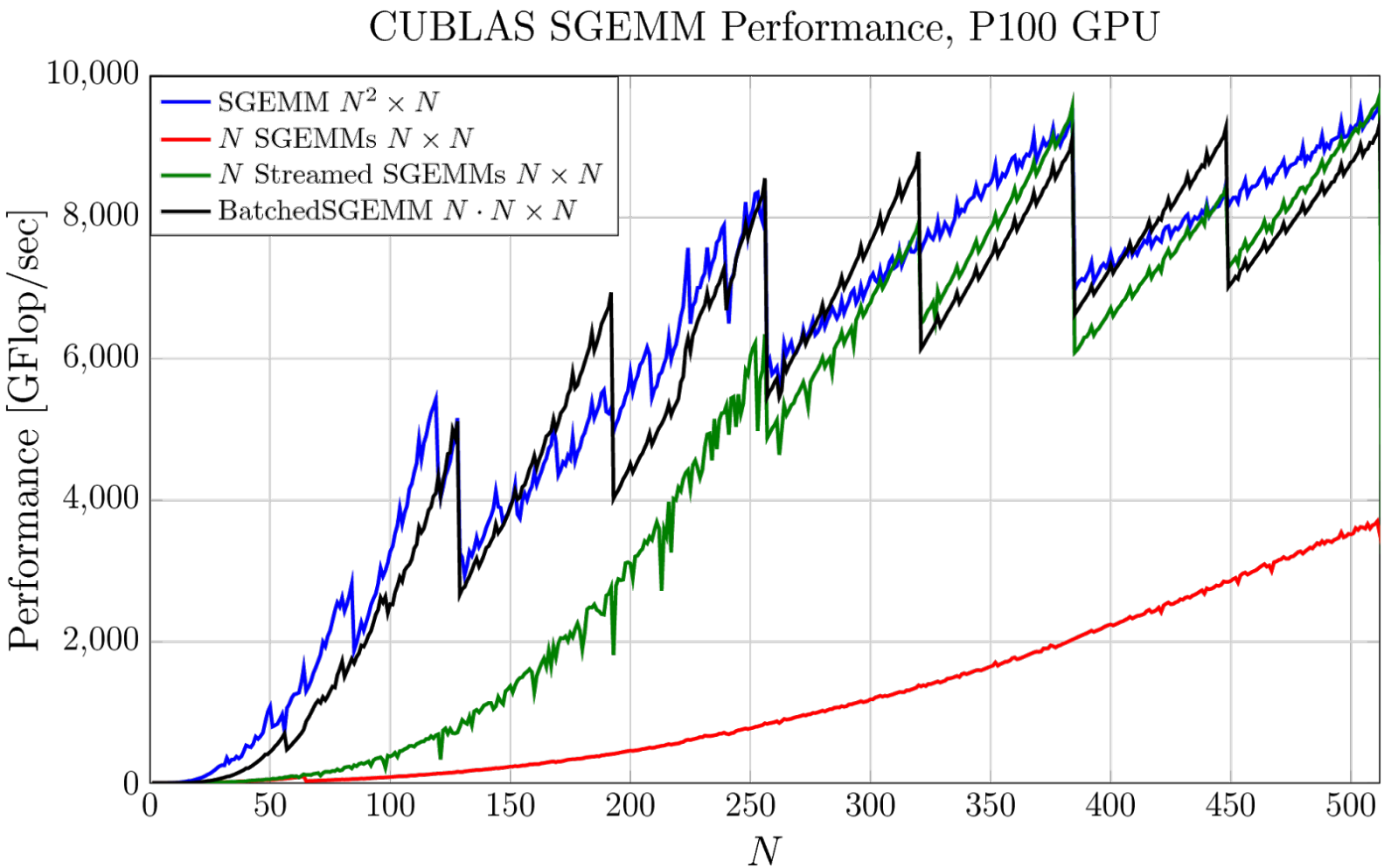

Image from https://developer.nvidia.com/blog/cublas-strided-batched-matrix-multiply/

From the figure, we see that a for-loop over SGEMMs (each matrix multiply is a kernel launch) faired the worst. Streaming (an option to asynchronously call multiple kernels) helps, but what is most beneficial is doing a batched multiply.
A typical lattice contraction can be considered as a batched BLAS call. Say I want to to

$$C(p,t) = \sum_x e^{ip\cdot x} \mathrm{Tr}\left[\psi(x,t)\bar{\psi}(0)\right] \tag{3}$$

Well, the Fourier transform over x is a vector-vector operation. If I have multiple values of p then this can be considered as a matrix-vector operation, and if I do the same for all t this becomes a matrix-matrix operation. If I want to do this now over all variations of ψ (noises, flavors... etc) that can be written as a batched matrix-matrix operation.

A BLAS call for the DFT could look like

$$N_P \overbrace{\left\{ \begin{bmatrix} e^{ip_0 \cdot x_0} & \cdots & e^{ip_0 \cdot x_{L^3}} \\ \vdots & \vdots & \vdots \\ e^{ip_{N_P-1} \cdot x_0} & \cdots & e^{ip_{N_P-1} \cdot x_{L^3}} \end{bmatrix} \right.}^{L^3} \times L^3 \overbrace{\left\{ \begin{bmatrix} Tr(x_0, t_0) & \cdots & Tr(x_0, t_{L_t-1}) \\ \vdots & \vdots & \vdots \\ Tr(x_{L^3}, t_0) & \cdots & Tr(x_{L^3}, t_{L_t-1}) \end{bmatrix} \right.}^{T} \tag{4}$$

This could be batched in N_P or T or ideally something bigger on the right hand side over noises/flavors etc..

Without doing a large number of momenta and giving the GPU a lot to do we will not beat a CPU implementation i.e. a zero-momentum 2pt contraction just over a timeslice will likely perform worse than a CPU implementation.

A call to cuBLAS does not need tuning for threads and blocks - they do that

To summarise

We learned that performance for simple operations on-chip is fantastic, but we can be limited by pushing the data to the GPU and getting it back.

QUDA interface code is just like CUDA's with memcpys and frees. QUDA has some nice templated objects for allocating and transferring memory to the device so that more concise code can be written.

There is quite a bit of boilerplate and obfuscation that comes with the autotuning when writing our own kernels. Partly this is needed for the templated support of multiple precisions too.

We introduced batched BLAS calls and showed in principle the performance benefits - this will be useful in the next lecture.